Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 8, Number 2, pages 98–103, July 2019

Siva – The IPFS Search Engine

Nawras Khudhur and Satoshi Fujita
Department of Information Engineering, Hiroshima Univetsity
Higashi-Hiroshima, 739-8527, Japan.
Email: {nawras, fujita}@se.hiroshima-u.ac.jp

(preliminary version)

***Abstract*—In the world of increasing data, it becomes necessary to upgrade the present structure of the network and go for the decentralized network. Therefore, IPFS (InterPlanetary File System) made it possible to make a pure Peer-to-Peer (P2P), censor resistant and permanent Web in which links will never die. Despite the merits of IPFS, finding data without knowing its unique name is impossible since there is no search engine to index the data and respond to queries. Our aim is to provide a decentralized search engine to IPFS in which the nodes can search for any content at any time and anywhere. we propose using DHT to handle the indexed data and strengthen the search with adding result cache layer to the search engine alongside with a hash filter to make cache lookup process fast and efficient. We used simulation to test the caching effect on the search engine using a sampled dataset from Yahoo Webscope. The simulation results show the effect of caching on reducing query latency and effect of different cache sizes on the cache hit-rate as the hit-rate decreases with decreasing cache size.**

***Index Terms*— DHT, p2p search engine, keyword searchable, cached query, hash filter.**

I. Introduction

The World Wide Web technology made a bold step towards a new level called Web 3.0 with introducing IPFS (InterPlanetary File System) which creates a pure Peer-to-Peer (P2P), censor resistant and permanent Web in which links will never die [1]. IPFS combines several proven P2P technologies such as distributed hash tables (DHT), BitTorrent, Git, and Self-certifying File System (SFS) [1] and creates a universal decentralized file system that overcomes the weaknesses of the current web such as a single point of failure. The existence of a single point of failure tends to make it easier for different entities to control the resource access by people as it occurred in Turkey when the government blocked access to Wikipedia on April 2017 [2]. At that time the activists published the Turkish version of Wikipedia on IPFS to avoid one point of failure. Thus, using IPFS it was possible to make Wikipedia uncensorable. Another important point to note is that, IPFS takes advantage of the fact that each different file produces unique hash to create a resource based network instead of location-based, that is, each file is recognized by its hash, not by the location that shares it, which helps to get the requested resource from all people that share this resource and simultaneously without putting the bandwidth burden on a single server.

Despite the fact that IPFS works great, currently, it doesn't support any kind of search through the P2P nodes except for an exact file hash look-up. To elaborate, the IPFS stores the *index* of files in a DHT that works as file availability DHT. The DHT keeps the key-value records of each file hash and the providers' address. In order for a user to find a file, He/She must know the exact hash of the requested file then ask IPFS to find the providers of the file to connect to them and download the data. The lack of search makes it difficult for users to search for files by its keywords or descriptions. Although this problem has not been tackled by anyone in the context of P2P systems, there is one project making attempts to create a generic search engine for the IPFS called **ipfs-search** [3]. ipfs-search is a centralized search engine based on the Elasticsearch that is a distributed, RESTful search and analytics engine [4]. ipfs-search crawls IPFS network by taking advantage of IPFS logs to monitor the *file add* events of other peers. This technique is called DHT sniffing, which means the ipfs-search server have the node(s) to monitor DHT record updates between their node(s) and others and catch the desired file add events. The crawler then catches the content identifiers (CID, explained in II-C) of added files and pushes them to some hash queue. The queue is processed to distinguish the files and directories, then the files are fetched and the result sent to Apache-Tika [5] for metadata extraction. Finally, the extracted information of the crawled files will be stored on Elasticsearch to be indexed and be available to search for the public. While the ipfs-search is still in its early stages, it provides users the ability to perform single/multiple keyword searches. However, having all indices on the server will lead to poor availability as well as scalability. For instance, when the server is attacked or receives a huge number of requests, it may overload and stop responding to the queries since the bandwidth is limited. Moreover, It opposes the aim of the IPFS that seeks distribution of data across the web not centralizing it. Admittedly, considering service to a significant number of nodes will make it very costly and less available due to a large amount of bandwidth that is required for most of the search engines, in line with that, the centralized search engine will limit the scalability of the IPFS. Having a robust and dependable search ability that is not affected by censorships or DDoS attacks is a must for the next web seeing that unavailability of the search for required data will limit the IPFS network in serving data to the restricted or low connectivity areas.

Our goal is to build a decentralized search engine for IPFS to help users query for the desired resources even when not knowing the exact hash of them. The first step is *indexing* the shared files by extracting and relating the keywords that describe each file then maintain such information in the p2p manner to be served for efficient query processing. There are differ-

ent ways to maintain such an index, as will be explained later, we propose using DHT to handle the indexed data that works independently on the *file availability DHT* used in IPFS. It's true that building a P2P search engine needs different fixes and techniques to become practical and efficient. One of the vital techniques to make query process rapid includes result cache. As stated by [6], [7], different caching techniques for multiple keyword searches reduces the bandwidth usage and also the latency in the P2P network considerably. In Kobatake et al [6] and Ariyoshi et al [7] the results show that the amount of returned data reduced by 60% and 49.7% respectively. Supporting the P2P searches using the result caching techniques will improve the latency and reduces the cost. In this paper, when a peer in the IPFS network issuing a query to search for some content, we draw on the DHT to respond to that query, meanwhile the requester node caches the list of results. Next time, when another node tries to issue a query, first, it will look into the cached result, if the query result was available, it returns the result without conducting an actual search through the IPFS network. Hence, we certainly increase the search result availability and reduce the latency.

II. IPFS Basics

A. Overview

An IPFS network consists of several autonomous **peers** with their own public and secret keys. Each of the peers is assigned a unique identifier called NodeId which is obtained by applying an appropriate hash function to the public key. In IPFS all peers can add **files** to the network to be shared with other peers. Each shared file is assigned another identifier called CID (Content Identifier) using the same hash function based on the **content** of the file. With the notion of identifiers, *content-based file retrieval* proceeds in the following manner: Initially, files are associated with peers in such a way that the index to a file is maintained by a peer to have the closest NodeId with the CID of the file. The requester uses the mentioned fact to find the responsible peers for holding indexing information about the requested file and then retrieves the information. After getting the index of a file from the responding peer, the retrieval of a file is conducted by forwarding a query message from the requester to peers which are likely to be associated with the requested file through an appropriate P2P overlay, as will be described later. Consequently, The index of the retrieved file updates as the requester will also become a seeder for the acquired content by automatically caching the file in a short-term manner. To deal with the limited cache, The unused cache will be garbage collected automatically by the peer itself. Alternatively, the garbage collection process can be initiated manually as well. In addition, the peer can exclude specific files from being garbage collected by *pinning* the file which makes the peer permanent provider for the pinned file. That means when a file is pinned, it's guaranteed that it won't get removed unless the pinner removes it manually.

B. Data Structure used in the IPFS

Using IPFS, users can share any kind of data like text, video, audio, even directories can be shared directly while keeping their tree structure. IPFS uses MerkleDag [8] to keep the full structure of files and directories. It is worth mentioning that MerkleDag is deprecated by now for the favor of InterPlanetary Linked Data (IPLD) [9] which is a more general data structure. IPLD is meant to behave the same as regular URL and links in the HTML.

In IPFS the files are located according to their content than their location and served by multiple nodes to the requester just like BitTorrent but in a single swarm. IPFS addresses items through content identifier CID, which is a cryptographic hash generated based on the content of the item. Any change to an item will result in a different CID and two identical items will produce the same CID. By this, IPFS can control the duplication and version of the items. Nevertheless, this behavior causes the CID addressing to be unstable and adds difficulties to share contents through IPFS network. To overcome this problem, IPFS provides a way to address files with a fixed link using Inter-Planetary Name System (IPNS) [1] that works similar to *Domain Name System*. By using IPNS, one can generate an IPNS link to the shared files and guarantee that the IPNS link will not change even when the file content changes. This technique makes it much easier to share the contents on IPFS.

C. Content Identifier (CID)

In IPFS each file and each block are given a unique fingerprint that corresponds to the cryptographic hash of that file/block and called content identifier or CID for short. The CID is 46 characters long self-describing content-addressed identifier, hashed using multi-hash protocol [10] that makes the used hash function and addressing size recognizable from the resulting hash by adding two leading bytes to the resulting hash. To illustrate, when a node adds a file to IPFS, first it generates the sha256 hash reflecting the content of the file, plus adding 2 leading bytes 0x12 and 0x20 that denotes the used hash function is sha256 and size is 32 bytes respectively (i.e. 12204660 DF5B7074A4E2C1DD7C07CC2EEA57C515F6E7A60E3FF0 16C3990FB48BA180 is the hash of Turkish Wikipedia version on IPFS), then this hash encoded in *base58btc* that results in 23 byte CID (i.e. QmT5NvUtoM5nWFfrQdVrFtvGfKFmG7AHE 8P34isapyhCxX).

When adding content to IPFS, the content will be stored differently according to its' size. If the content is equal to or less than 1KB then its directly stored in the DHT otherwise, the content is divided into smaller blocks (256k each) and the DHT stores references, which are the NodeIds of peers that can serve the block [1]. In another way, CID is used as a key in the DHT key-value store of IPFS.

Although the added item is still stored locally by the owner peer, advertised throughout the network using the DHT. Hence, when any node requests a specific CID, in the backend the requester node will query the DHT for that CID to get NodeIds that are seeding the requested content, then connections are established to the responsible NodeIds and the content is transferred in blocks of data to the requester.

The blocks of data are exchanged with Bitswap protocol [1] that is inspired by the BitTorrent protocol. The key difference here from BitTorrent is that in BitTorrent the blocks being ex-

changed are all from a single torrent, while on IPFS there is one big swarm of all IPFS data.

D. IPFS layers

The IPFS layers as shown in Figure 1, divided into three main layers which are Moving data, Defining data and Using data [11]. Moving data layer is called libp2p which is a networking layer of IPFS. It consists of a collection of different P2P protocols and modules Figure 2a 2b.

libp2p consists of three sub-layers namely: **network**, **routing** and **exchange** layers. The network layer provides reliable and unreliable point-to-point connections between any two IPFS nodes within the network, the routing layer provides peer routing and content routing to find other nodes and requested data on the network respectively. The IPFS Block Exchange layer manages the transfer of the blocks of data among nodes.

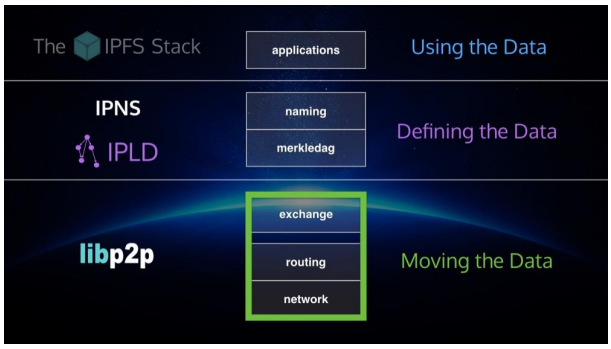

Fig. 1. IPFS layers [11]

III. Proposed Solution

The proposed solution as shown in Figure 3 is to build an inverted index of keywords called **search-index** for the published contents on IPFS and give the search ability to users.

The search-index is a list of key-value pairs that connect keywords to the published contents as ¡keyword, CID¿ pairs. While text search is a query of a single/multiple keyword(s) that may match with the extracted information of the files that are shared over IPFS. The matching result is the list of *CIDs of files* which contain the queried keyword(s). In case of multiple keywords query, searches will include some set operations over

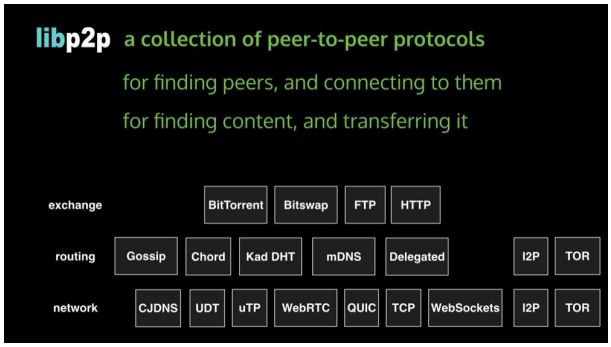

Fig. 2a. IPFS libp2p [11]

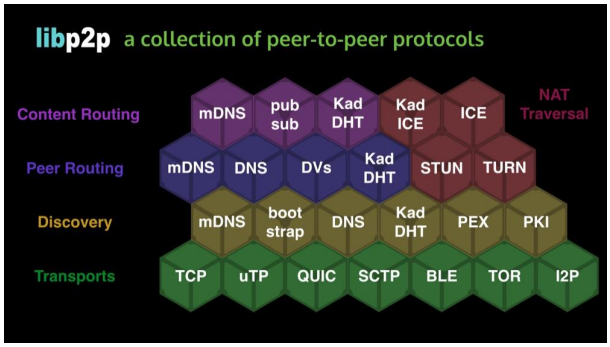

Fig. 2b. IPFS libp2p [11]

the single keyword results. For example for query $Q = A \wedge B$ where $A, B \in \{keywords\}$, the final result is calculated as $R_{A,B} = R_A \cap R_B$

Adding a result cache layer between the IPFS peers and the search engine as shown in Figure 4 will make the searches more efficient. that means when a peer makes a search, first, the search engine looks for the cache, if the result for that query was cached then it is fetched by the requester. Otherwise, the requester sends the query to the search-index. To further extend the efficiency of the search engine, we add a hash filter layer to give a faster answer about the existence of the issued query result cache.

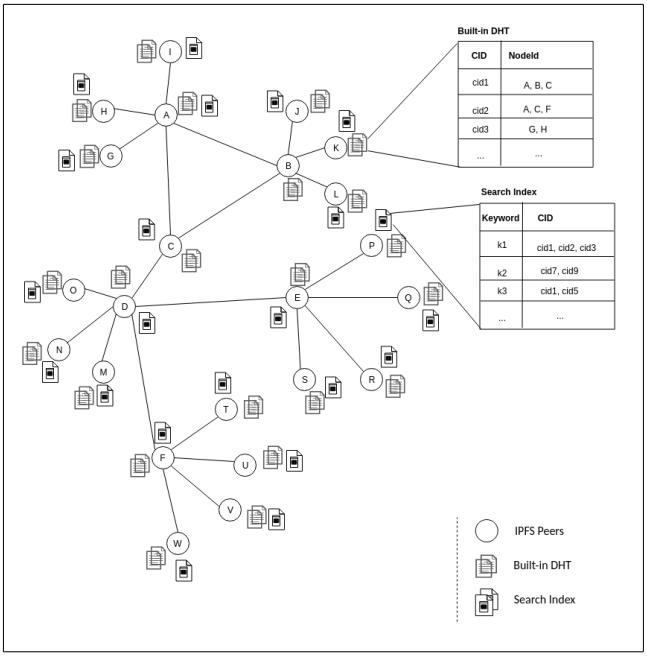

Fig. 3. Network Architecture

A. Architecture

The architecture of the proposed search engine consists of the followings:

- The IPFS peers
  The IPFS peers make up the P2P network.
- IPFS search engine
  Serves the main keyword index (search-index) to IPFS

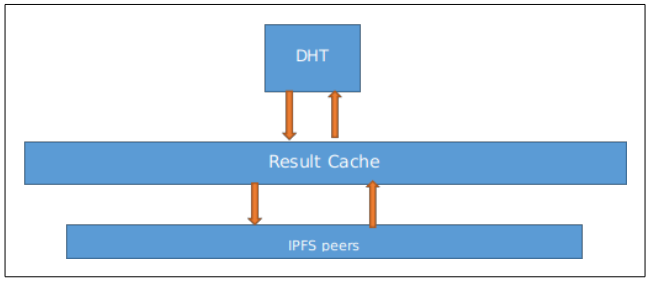

Fig. 4. Result Cache Structure

peers and it returns the result of the query (list of CIDs) by looking up the indexed data.

- Result cache layer
  The responsible layer for caching query results of common queries and serves it again to the peers.
- Hash filter layer
  Increases the efficiency of finding the caches for the requested queries.

B. Method

Each and every peer will participate in building and maintaining the inverted index for the files shared over IPFS. The inverted index called search-index is stored in a DHT separate from the built-in routing DHT. Using the separate DHT than built-in routing DHT lets our search engine implementation be flexible and easy to adapt to any changes on IPFS. Besides, each node stores a portion of the search-index that is closer to their NodeIds. Additionally, we use result caching techniques to make the searching process more efficient and less traffic hungry. To further clarify, when a node adds a file to IPFS, it can choose whether to **index and add** or just **add** the file. In the case of index and add, the file goes through a set of processes to be indexed then added. The steps are explained as:

1) Metadata Extraction: In the first step of adding a file to IPFS, the file is sent to a metadata extractor such as Apache Tika [5] to extract the basic information out of the file.

2) Keyword Extraction: After metadata extraction, if the added file is a text file, the text content is analyzed using TF-IDF in order to weight the keywords. Thus the rare keywords are chosen to represent the document.

3) Building The Inverted Index: When the metadata and the weighted keyword list is generated, the CID of the processed file with its list of representative keywords and metadata is sent to the corresponding peers to be inserted/updated in their search-index DHT portion.

As a result, the added file is indexed and available to query according to its' extracted data.

C. Cache generation

In principle, when any peer in the IPFS network makes a query search, the hash of the query and a part of the result is stored in a form of database file (such as OrbitDB [12] which is a key-value store, serverless, distributed, peer-to-peer database that uses IPFS as its data storage) by the issuer node, then it will be served again to other peers who query for the same keywords, that means the query issuer is responsible for caching

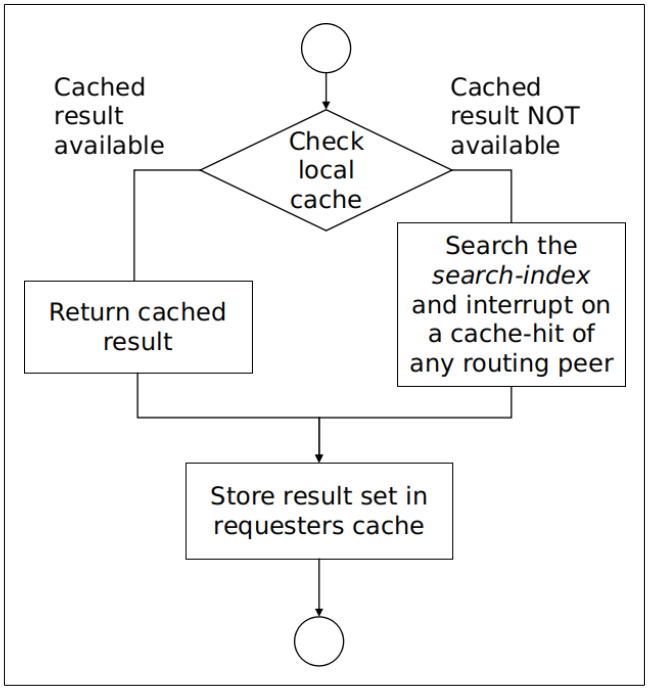

Fig. 5. Search process & cache generation flow chart

the results, see Fig 5. So, the process is as following: when a peer issues a query search Q, at first it will check the cached data for the requested query, if the query result was cached by the node itself then the result will be returned to the user without proceeding to the text search, otherwise start initiating the search request to search-index but during the routing of the query Q, if any routing peers happened to have the required result in their cache, interrupt the searching and return the result. If there were no cached results for query Q among the routing peers, then the query Q will eventually reach the peers who are responsible for the corresponding key part of the search-index, then the actual result will be returned.

The cache is stored in the key-value form, the key is the query term(s) and the value is the text result of the query search. Since the cache storage is limited and query results always get cached once it's issued, when cache storage is full the cache replacement algorithm will be applied to push the new cached results into the result cache storage.

D. Hash Filter on Cache Results

Hash filters are used as membership testing against a set of the existing data. Hash filter can respond by ensuring that the item is not a member of the set or the item is probably a member of the set, such probability is called false positive probability.

Cuckoo filter is a new data structure, represented in a paper by Fan et al in 2014 [13]. Cuckoo filters improve upon the scheme of the bloom filter by giving deletion, restricted counting, and a bounded false positive probability, whereas still keeping up a similar space complexity. Customarily, the cuckoo filter uses two hash functions and two layers of arrays to store the hash of the item, if the bucket in the first layer was empty then push the value into it otherwise push the value into the second layer.

Because our cache result is not static, which means it will be updated or deleted after some period of time, we need to have a hash filter that can support item deletion. The cuckoo hash filter provides such an ability. So, our search algorithm will be changed as the following:

- User x makes a query search for some query Q.
- The cuckoo filter used to find out if the query Q exists in the cached result or not.
- If query result was cached, return it to the requester.
- otherwise, proceed to the text search on the search-index

IV. Simulation Model

The evaluation of the proposed idea is done through simulation using Peersim [14]. A DHT is used to simulate the environment and search-index distribution over IPFS. The P2P environment set to be static, which means the state of peers doesn't change during the simulation. We used sampled dataset from Yahoo! Search Engine provided by Yahoo! Webscope [15]. The dataset contains 1500000 queries in total with 1198 unique queries, see Fig 6.

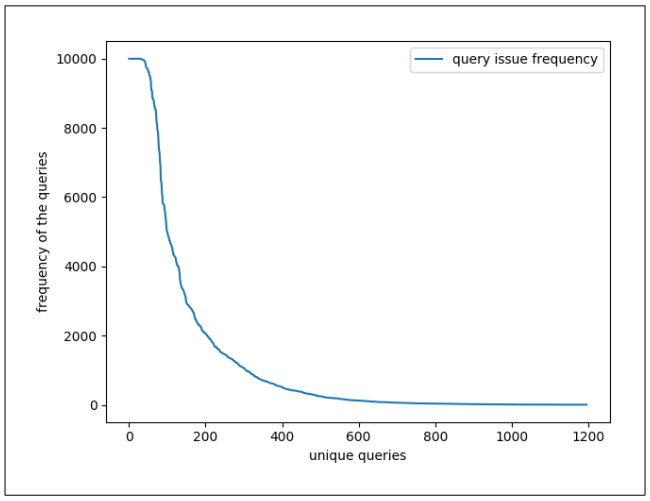

Fig. 6. Yahoo! Webscope dataset

The population of the search-index is done by processing the dataset and put it in the search-index DHT, the search-index is distributed according to the used DHT algorithm (Kademlia [16] used in this simulation) and each peer in the network is responsible to maintain a part of search-index. The query results are cached only if the query was answered before, and it will be stored according to the used DHT algorithm. We also consider the query locality [17] to increase the caching performance. As the cache capacity is limited, the query results will not be cached completely rather, the first few results will be cached. In addition, the cache will eventually expire according to a cache replacement scheme (We have used least recently used scheme in this simulation) to prevent memory overflow. To compare and see how much cache capacity is needed to ensure the reasonable hit rate, the cache size will vary and will be predetermined through parameters.

The simulation started with peers sharing no data and empty search-index, they gradually built their search-index, then the search process issued 10100 queries over 3 hours of simulation time, and cache size starts from 0 unit to 20 units, each unit represents storage for one query result. Queries were chosen randomly but influenced by the frequency of the queries in the dataset. We tested different settings of cache size from 0 to 20 units adding 5 units each time for two different network size 500 nodes and 1000 nodes. The results show that adding cache to the search process makes queries take less time on average, from 1870ms per queries when 0 cache, to 1476ms when 20unit of cache is used, which is about 22% reduction. Additionally, the cache hit-rate also increased from 0 to 3372 total hits, see Fig 7a. on the other hand cache seems less effective when network size grows higher as shown in Fig 7b that queries take 1937ms on average when 0 cache is used and reduced to 1696ms for 20 unit of cache, which is about 12% reduction in time, and cache hit increased from 0 hits to the total of 2003 hits.

Network Size 500		
cache size	avg query msg time	cache hit
0	1870	0
5	1561	2407
10	1504	3007
15	1495	3159
20	1476	3372

Fig. 7a. Simulation Results - Network Size 500

Network Size 1000		
cache size	avg query msg time	cache hit-rate
0	1937	0
5	1708	1785
10	1688	2027
15	1686	2001
20	1696	2003

Fig. 7b. Simulation Results - Network Size 1000

V. Conclusion

We showed the effect of caching on reducing query latency and effect of different cache sizes on the cache hit-rate as the hit-rate decreases with decreasing cache size.

Since multiple keyword searches create more burden on the network and result caching is more effective in such environment, We have to test the multiple keyword searches as well as applying the hash filter layer to the search engine and compare the cache effectiveness with current results.

References

[1] J. Benet, "Ipfs-content addressed, versioned, p2p file system," *arXiv preprint arXiv:1407.3561*, 2014.
[2] Turkeyblocks, "Wikipedia blocked in turkey," https://turkeyblocks.org/2017/04/29/wikipedia-blocked-turkey/, accessed: November 26, 2018.

[3] M. de Bruin, "Search engine for the interplanetary filesystem," https://github.com/ipfs-search/ipfs-search, accessed: October 10, 2018.
[4] "Elasticsearch," https://www.elastic.co/products/elasticsearch, accessed: October 10, 2018.
[5] "Apache tika - a content analysis toolkit," https://tika.apache.org/, accessed: October 12, 2018.
[6] K. Kobatake, S. Tagashira, and S. Fujita, "A new caching technique to support conjunctive queries in p2p dht," *IEICE - Trans. Inf. Syst.*, vol. E91-D, no. 4, pp. 1023–1031, Apr. 2008. [Online]. Available: http://dx.doi.org/10.1093/ietisy/e91-d.4.1023
[7] T. Ariyoshi and S. Fujita, "Efficient processing of conjunctive queries in p2p dhts using bloom filter," in *International Symposium on Parallel and Distributed Processing with Applications*, Sept 2010, pp. 458–464.
[8] "The merkledag," https://github.com/ipfs/specs/tree/master/merkledag, accessed: October 23, 2018.
[9] "Ipld," https://github.com/ipld/ipld, accessed: October 23, 2018.
[10] "Multihash," https://github.com/multiformats/multihash, accessed: October 23, 2018.
[11] "Libp2p," https://github.com/libp2p/libp2p, accessed: October 23, 2018.
[12] "Orbit-db," https://github.com/orbitdb/orbit-db, accessed: October 12, 2018.
[13] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14. New York, NY, USA: ACM, 2014, pp. 75–88. [Online]. Available: http://doi.acm.org/10.1145/2674005.2674994
[14] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," in *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, Seattle, WA, Sep. 2009, pp. 99–100.
[15] "Yahoo! webscope dataset anonymized yahoo! search logs with relevance judgments version 1.0," http://labs.yahoo.com/Academic_Relations.
[16] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, ser. IPTPS '01. London, UK, UK: Springer-Verlag, 2002, pp. 53–65. [Online]. Available: http://dl.acm.org/citation.cfm?id=646334.687801
[17] P. J. Denning, "The locality principle," *Commun. ACM*, vol. 48, no. 7, pp. 19–24, Jul. 2005. [Online]. Available: http://doi.acm.org/10.1145/1070838.1070856